Executive Summary

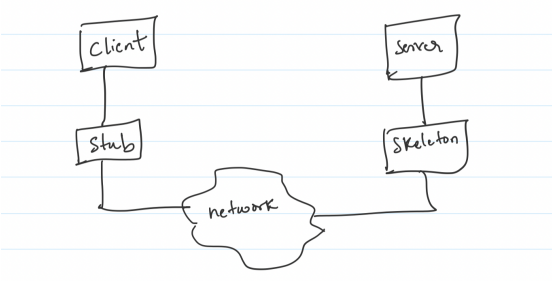
Assignment Overview:

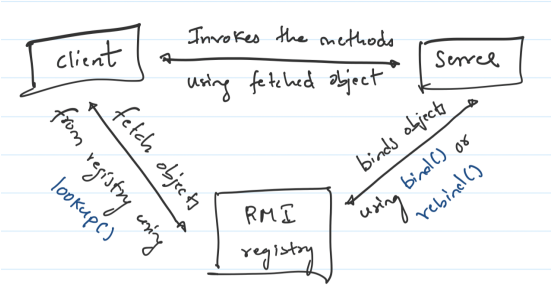
The assignment aims to develop a multi-threaded key-value store system with communication facilitated through Remote Procedure Calls (RPC) such as Java RMI (Remote Method Invocation). The server should be multi-threaded and should be able to handle multiple clients to perform various HTTP-like operations. The HTTP operations like PUT, GET, and DELETE operations need to be supported by the key-value store application. A map can be used to store the key-value pairs. As the operations are performed concurrently, mutual exclusion needs to be handled at the server. The scope of the project includes designing and implementing the server-side functionality to handle the above operations while ensuring mutual exclusion to prevent data inconsistency. The client must also pre-populate the key-value store with data, conduct 5 PUT, 5 GET, and 5 DELETE operations, and communicate with the server via RMI for these actions. Thus the assignment aims to demonstrate the implementation of distributed computing, multi-threading, and concurrent access control to build a robust, concurrent key-value store using Java RMI.

Technical Impression:

I have learned the following during my implementation of the project:

Java RMI uses RMI Registry to keep track of the server objects. The objects at the server need to be registered with the registry using the bind() or rebind() method. To communicate with the server, the client needs to know the remote reference of the server object which it fetches from the registry using the lookup() method. The client then uses stub which acts as a proxy to the remote object to communicate with the server. The server side uses a skeleton to communicate with the client. The skeleton then passes the request to the object on the server side. The reverse flow occurs once the server processes the client requests.Java RMI internally supports multi-threading. The server spawns new threads to handle the incoming requests from the clients automatically.





In my application, the registry has been integrated into the server. It can also be run as a separate process as well if needed.

I have used Java's Reentrant Read Write locks to handle the mutual exclusion. As the GET/PUT/DELETE operations can be performed simultaneously, this could lead to a classic "Critical Section Problem". To protect these concurrent operations from each other, I have used read, and write locks. For GET operation and map.containsKey() methods, I have used read locks so that if concurrent reads are happening, the read locks can be obtained as the data is not affected by concurrent reads. But while the read is happening, concurrently no other thread can modify the current object. For the PUT/POST/DELETE operations, I have used write locks so that when the addition or deletion of keys is performed by one thread, it does not coincide with other threads reading, adding, or deleting the same object. Another approach could have been to use ConcurrentHashMap.

Adding on to the project 1 logging, the current thread is also logged with its thread ID to trace the threads.

````
# key-value-store


### Executable


```

keystore.jar
```


### Program Arguments


```

-server    -> run as server

-client    -> run as client

-port      -> server rmi port
````
````

```

### How to run server?

```
java -jar keystore.jar -server   -> defaults to port 1099
java -jar keystore.jar -server -port 8081
```

### How to run client?

```
java -jar keystore.jar -client                     -> defaults to server rmi port 1099
java -jar keystore.jar -client -port 8081
```

Log files location

```
user home dir
%h/client%u.log     -> client logs
%h/server%u.log     -> server logs
```

Commands
```

```

post <key> <value>

put <key> <value>

get <key>

delete <key>

Application also pre-populates 10 post entries, 5 put entries, 5 gets and 5 deletes from key_value.txt file

```